

Solution to Himsord's 'Infinity Keygenme v1'

After some abstinence in reverse engineering I thought it might be time to get my hands back on assembler and so I looked through the last submits on crackmes.de and stumbled upon this one. It looked interesting to me and so I gave it a try...

About this crackme

URL	http://crackmes.de/users/himsord/infinity_keygenme_v1/
Author	Himsord
Published	25. Jan, 2016
Platform	Windows
Language	C/C++
Difficulty	2 - Needs a little brain (or luck)
Protection	System-Date Check

Tools used

- OllyDbg v2.01
 - Visual Studio 2015
 - stackedit.io (for writing this tutorial)
-

First analysis

We open up the program and it asks us for an ID and a password. So we enter some random data like **123456** and **abcdef**. Of course it gives us the badboy: 'You didn't crack me.' Time to start Olly and solve this one ;)

Debugging

Attention: Please note that the program was compiled using the /DYNAMICBASE flag, so all addresses are random! You'll most likely see other addresses in your debugger!


So we open up the program in Olly and do the classic *Rightclick > Search for > All referenced strings* and come up with this:

```
00281B08 MOV EDX, OFFSET 002845F4 "You cracked me."
00281B1D MOV EDX, OFFSET 00284604 "You didn't crack me."
```

This really looks like the good/badboy!

Doubleclick on one of the strings and you land directly in the code:

```
00281B00 CALL 00281770
00281B05 ADD ESP, 30
00281B08 MOV EDX, OFFSET 002845F4 ; ASCII "You cracked me."
00281B0D TEST EAX, EAX
00281B0F PUSH 002823B0
00281B14 PUSH ECX
00281B15 MOV ECX, DWORD PTR DS:[<&MSVCP120.?cout@std@3V?$basic_ostream@DU?$
00281B1B JNZ SHORT 00281B22 ; Badboy is jumped
00281B1D MOV EDX, OFFSET 00284604 ; ASCII "You didn't crack me."
00281B22 CALL 00282170
```

First we have a call to an (yet) unknown subroutine, then a pointer to the goodboy is loaded into EDX. EAX is then tested and if it is zero, the badboy is loaded into EDX instead. So for now we can conclude, that whatever happens inside of the CALL sets EAX. Set a breakpoint on that CALL and start the program via .

Mhhh, but what is happening? We get some access violations and the program won't start. Moreover we can't pass the exception to the program, there must be some anti-debugging protection going on!

There are a lot of debugging-detection methods, but clearly the easiest one is to ask Windows if the program is debugged via a call to *IsDebuggerPresent*¹. To find out if the program uses this function we reload the program, right click the code and then select *Search for > All intermodular calls*. Now we start typing in the first letters 'ISDE' and should land here:

```
00171962 CALL DWORD PTR DS:[<&KERNEL32.IsDebuggerPresent>]
```

We double-click this and land in a very interesting routine starting at 00171920. Set a breakpoint at the beginning of the routine and start the program. The first CALL inside of the routine we break in is a call to *FindWindowW*² which

‘Retrieves a handle to the top-level window whose class name and window name match the specified strings.’

The specified class name is “OLLYDBG”, so the program wants to find out if OllyDbg is running and since we are using Olly it will return a valid handle. So it looks like we found the debugging-detection subroutine :D I didn’t really dig deeper into the other functions since I don’t know all the anti-debugging tricks, but thankfully the author of this crackme made it quite easy to get past all this anti-debugging stuff by simply returning immediately at the start of the routine. To do this, double-click on *PUSH ESI* at the start of the routine and write ‘RET’ into the appearing input box. A patched version of the program is included in the solution folder.

So now that we have dealt with the anti-debugging, it’s time to deal with the password-checks. Let the program run freely and enter a random ID and a random password. Once you have done that you should finally break on the CALL to 001251770.

Step into the CALL via and trace a bit through the code. Via entering different IDs and observing the code I found out that the first interesting code is at 013117DE:

```
013117DE CMP DWORD PTR SS:[EBP-38], 6
013117E2 JE SHORT 013117FF ; strlen(id) == 6
```

Here the program checks if our entered ID has a length of exactly 6 chars. Going a bit down the code we find the following:

```
013117FF PUSH 0
01311801 CALL DWORD PTR DS:[<&MSVCR120._time64>] ; \MSVCR120._time64
01311807 ADD ESP, 4
0131180A MOV DWORD PTR SS:[EBP-30], EAX
0131180D LEA EAX, [EBP-30]
01311810 MOV DWORD PTR SS:[EBP-2C], EDX
01311813 PUSH EAX
01311814 CALL DWORD PTR DS:[<&MSVCR120._localtime64>] ; \MSVCR120._localtime64
0131181A MOV ESI, DWORD PTR DS:[<&MSVCR120.??3@YAXPAX@Z>]
01311820 ADD ESP, 4
01311823 MOV EDX, DWORD PTR DS:[EAX+14] ; = YEAR (1990-based)
01311826 MOV ECX, DWORD PTR DS:[EAX+10] ; = MONTH (zero based)
01311829 ADD EDX, 76C ; YEAR += 1900
0131182F MOV EAX, DWORD PTR DS:[EAX+0C] ; = DAY
01311832 INC ECX ; MONTH += 1
01311833 SUB EDX, ECX ; YEAR -= MONTH
01311835 ADD EDX, EDX ; YEAR *= 2
01311837 ADD EAX, EDX ; DAY += YEAR
01311839 IMUL EAX, EDX ; DAY *= YEAR
0131183C IMUL EDI, EAX, 539 ; DAY *= 1337
```

First the program retrieves a UNIX-timestamp via calling *_time64*, then transforms this value into a localtime

value. After that it takes the year, month and day and calculates the following formula:

$$(Day + (Year - Month) * 2) * ((Year - Month) * 2) * 1337$$

So for example if we take the 30th January 2016, we would get:

$$(30 + (2016 - 1) * 2) * ((2016 - 1) * 2) * 1337 \mod 2^{32} = 21875726600 \mod 2^{32} = 400890120$$

Note that we are working with 32Bit registers, so we need to calculate the value mod 2^{32} to get the value as the program sees it.

Again, follow the code a bit and with a bit blackboxing you'll find this:

```
01311867 CALL int2hexStr
0131186C MOV BYTE PTR SS:[EBP-4], 2
01311870 LEA EAX, [EBP+20]
01311873 CMP DWORD PTR SS:[EBP+34], 10
01311877 PUSH DWORD PTR SS:[EBP+30] ; /Arg4
0131187A CMOVAE EAX, DWORD PTR SS:[EBP+20] ; |
0131187E PUSH EAX ; |Arg3
0131187F PUSH DWORD PTR SS:[EBP-18] ; |Arg2
01311882 PUSH ECX ; |Arg1
01311883 LEA ECX, [EBP-28] ; |
01311886 CALL strcmp ; \Infinity.strcmp
```

There are two calls here, one at 01311867 and the other at 01311886. As you can see, I already named them for you. The first one takes our calculated value and returns a hexadecimal string representation of that value (in uppercase). The second one simply takes that string and our entered password and compares them.

That is it ;) All we have to do is take the formula, calculate the value and convert it into an hexadecimal string, so for our date (2016-01-30) that would be the decimal value 400890120 and the hexadecimal string **17E51908**. For the ID you can enter whatever you want as long as it is 6 chars long!

Writing a keygen is pretty straightforward so I won't comment my solution here ;)

Summary

This was a pretty easy crackme, good to get back into reading ASM :D

(c) iSSoGoo, 2016

Footnotes

1. Read more about **IsDebuggerPresent** at [MSDN](#). ↩
2. Read more about **FindWindowW** at [MSDN](#). ↩